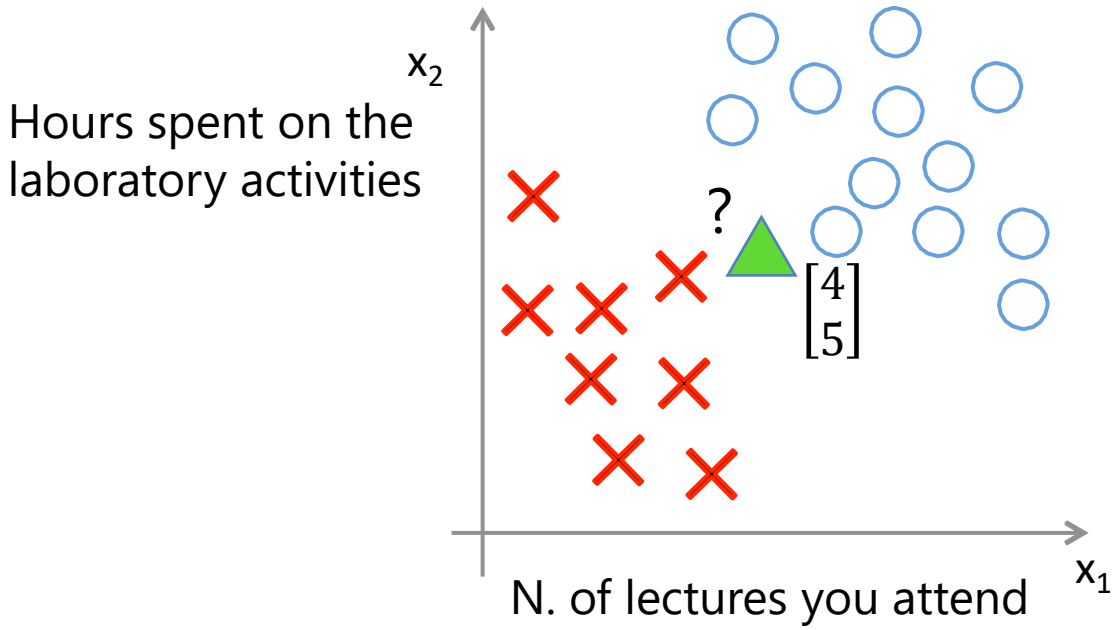
Neural Networks – Part2

Prof. Giuseppe Serra

Loss Functions

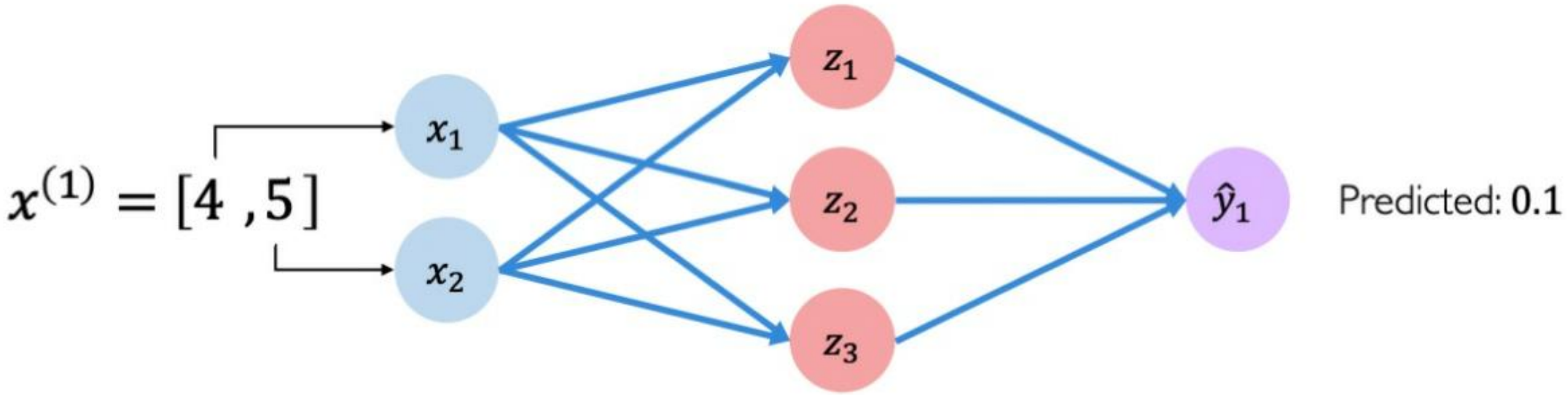
Example Problem: Will I Pass this Class?



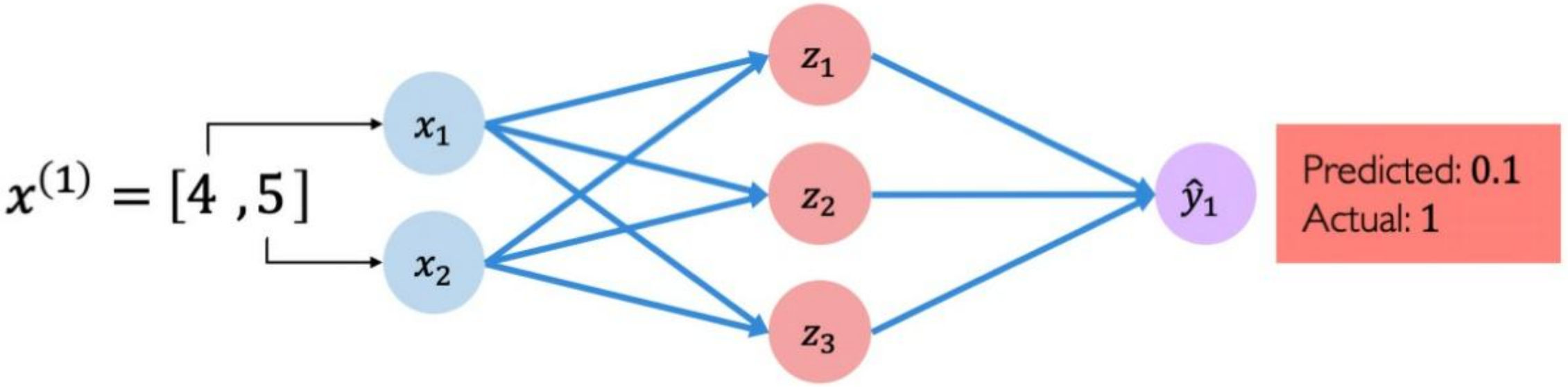
O corresponds to “Pass”

X corresponds to “Fail”

Example Problem: Will I Pass this Class?

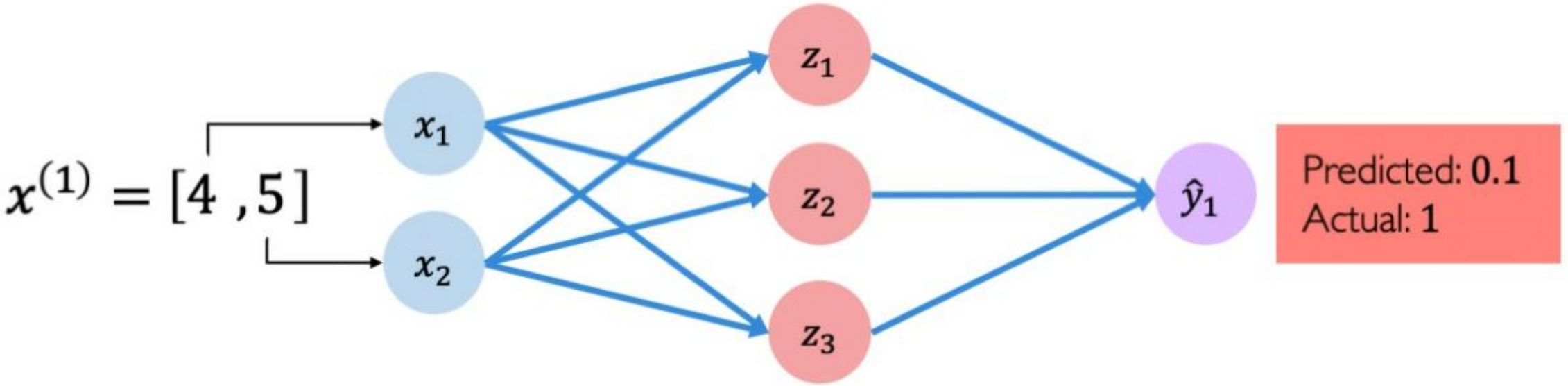


Example Problem: Will I Pass this Class?



Quantifying Loss

The Loss of our network measures the cost incurred from incorrect predictions

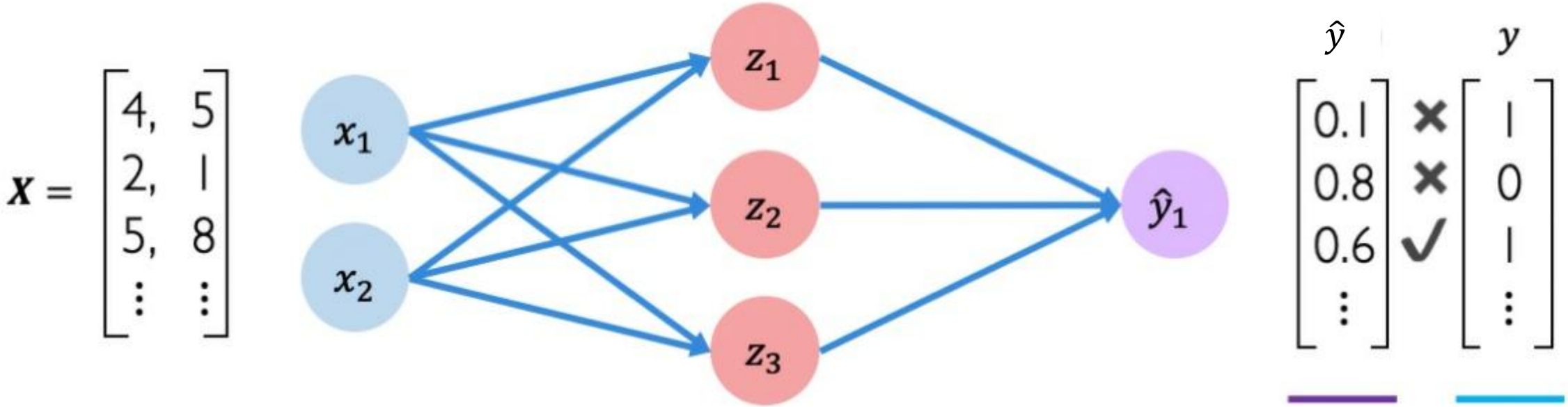


$$\mathcal{L}(\hat{y}, y)$$

Predicted Actual

Empirical Loss

The **empirical loss** measures the total loss over entire dataset

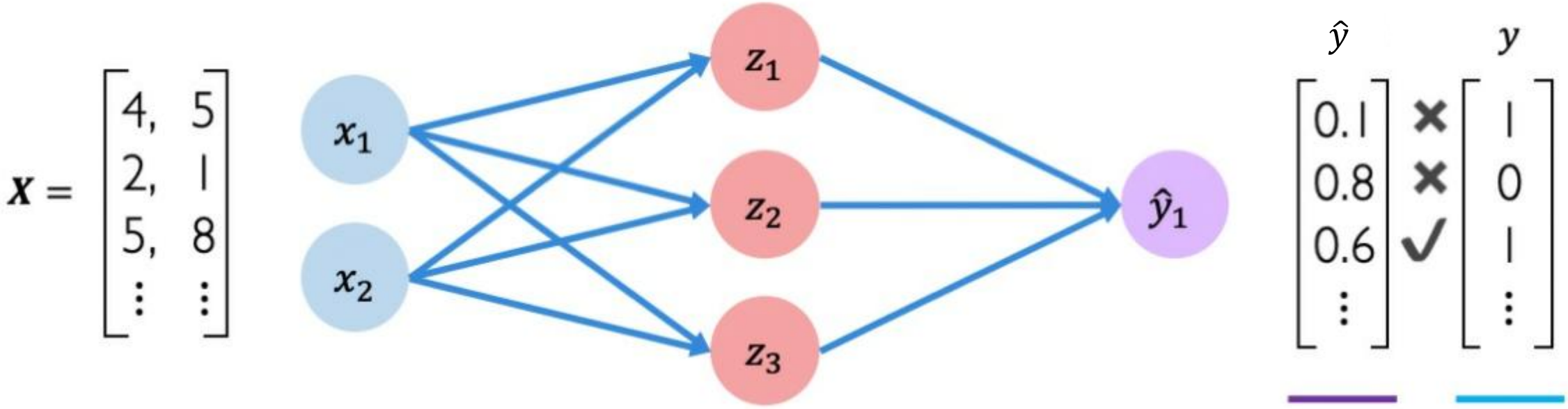


Also known as:
- Objective function
- Cost function
- Empirical Risk

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Binary Cross Entropy Loss

Binary Cross entropy loss can be used with models that output a probability between 0 and 1.



$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} -\left(y^{(i)} ln\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right) \ln\left(1 - \hat{y}^{(i)}\right)\right)$$

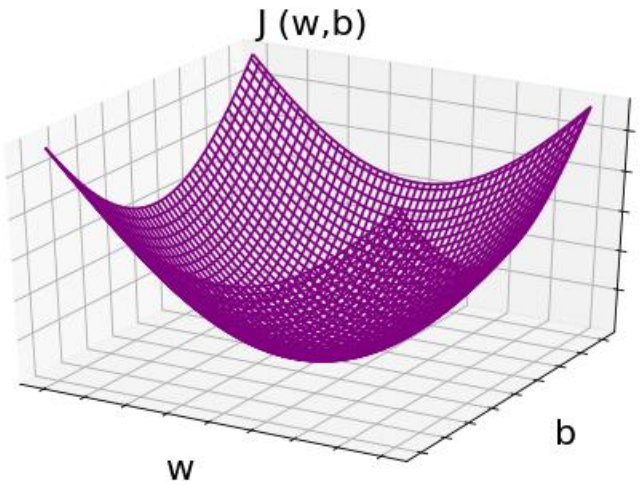
Cost Function

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ \ldots, (x^{(m)}, y^{(m)}),\}$

To find w and b so $\begin{cases} h_{w,b}(x) \geq 0.5 & if \ \ y = 1 \\ h_{w,b}(x) < 0.5 & if \ \ y = 0 \end{cases}$

We can define the following cost function:

- $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}\ (h_{w,b}(x^{(i)}), y^{(i)})$
  where $\text{Cost}\ (h_{w,b}(\ x^{(i)}), y^{(i)}) = \ -y^{(i)} \ln\left(h_{w,b}(x^{(i)})\right) - (1 - y^{(i)}) \ln\left(1 - h_{w,b}(x^{(i)})\right)$



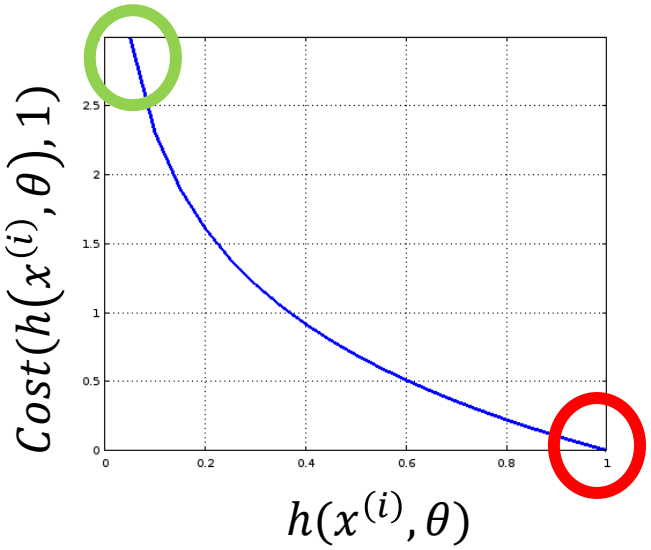
Note: This cost function (or loss) is called Binary Cross Entropy; it is convex and derivable respect to w and b

Cost Function

$$\text{Cost}\left(h_{w,b}\left(x^{(i)}\right), y^{(i)}\right) = -y^{(i)} \ln(h_{w,b}(x^{(i)})) - \left(1 - y^{(i)}\right) \ln\left(1 - h_{w,b}\left(x^{(i)}\right)\right)$$

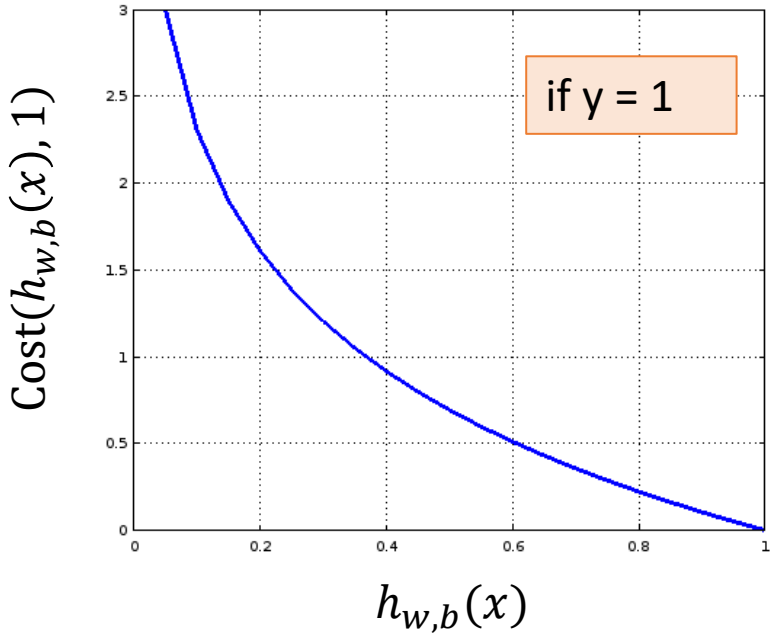
$$\text{If } y^{(i)} = 1 \Rightarrow \text{Cost}\left(h_{w,b}\left(x^{(i)}\right), y^{(i)}\right) = -\ln\left(h_{w,b}\left(x^{(i)}\right)\right)$$

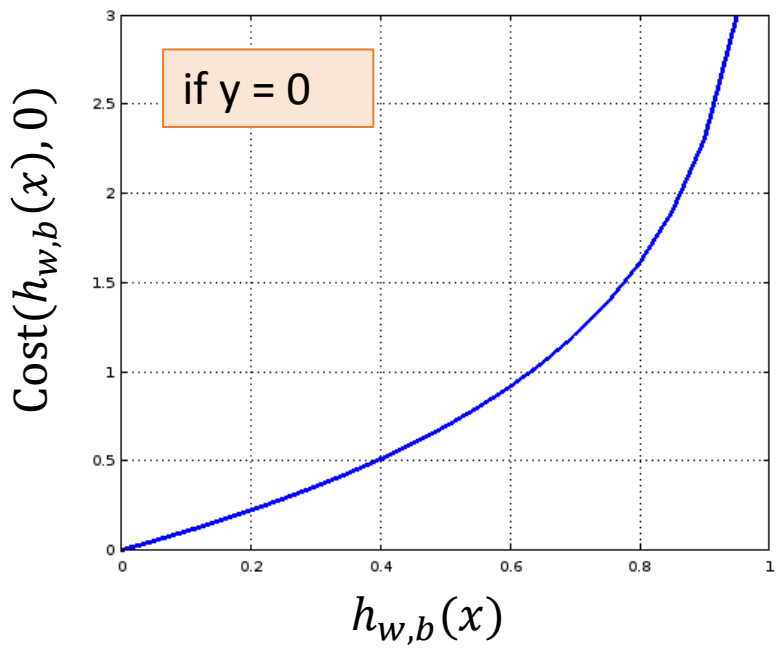
True Label $y^{(i)}$	Prediction $h_{w,b}(x^{(i)}, \theta)$	Cost $\text{Cost}\left(h_{w,b}\left(x^{(i)}\right), y^{(i)}\right)$
1	~1	~0
1	~0	Inf



Cost Function

$$\text{Cost}\left(h_{w,b}(x^{(i)}), y^{(i)}\right) = -y^{(i)} \ln(h_{w,b}(x^{(i)})) - \left(1 - y^{(i)}\right) \ln\left(1 - h_{w,b}(x^{(i)})\right)$$

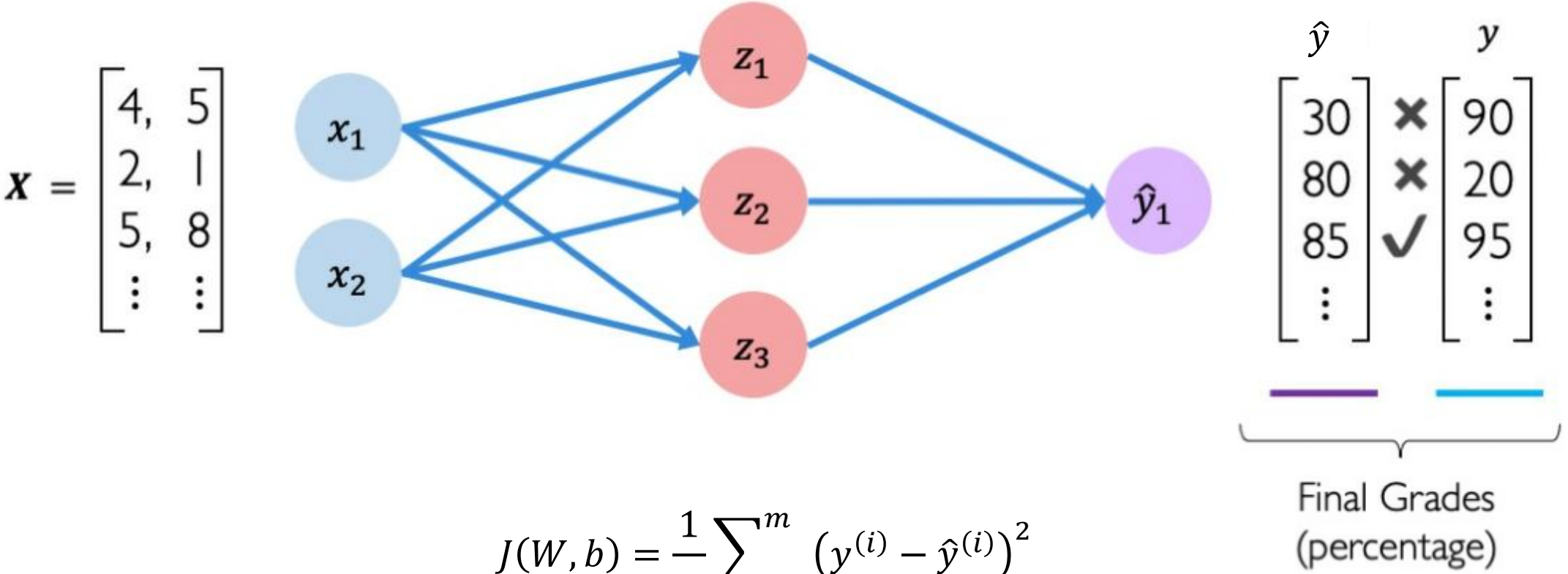




Case y=1, the cost function will be 0 if our hypothesis function $h_{w,b}(x)$ outputs 1. if our hypothesis approaches 0, then the cost function will approach infinity.

Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers.



$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

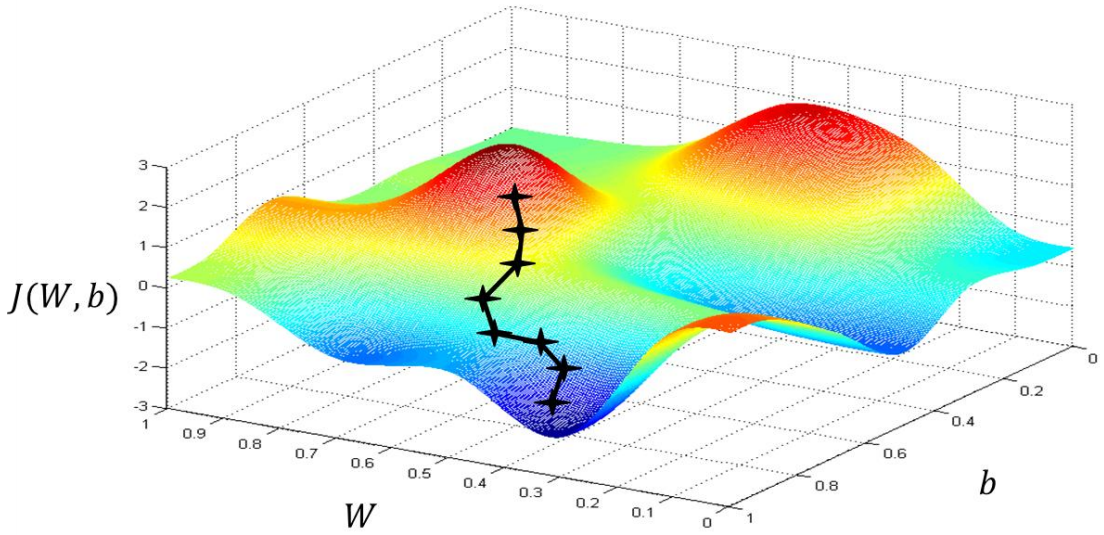
Training Neural Networks

Loss Optimization

We want to find the network weights and biases that achieve the lowest loss

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

$$[w^*, b^*] = \min_{W,b} J(W,b)$$



Gradient Descent

Gradient Descent Algorithm

GD is a general algorithm to minimize derivable function. Here we are using to linear regression.

Have some function $J(\theta_0, \dots, \theta_n)$

Want $\min_{\theta_0, \dots \theta_n} J(\theta_0, \dots, \theta_n)$

Outline:

- Start with some $\theta_0, \dots, \theta_n$ (common choice: random)
- Keep changing $\theta_0, \dots, \theta_n$ to reduce $J(\theta_0, \dots, \theta_n)$
- Until we hopefully end up at a minimum

Gradient Descent Algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$

}

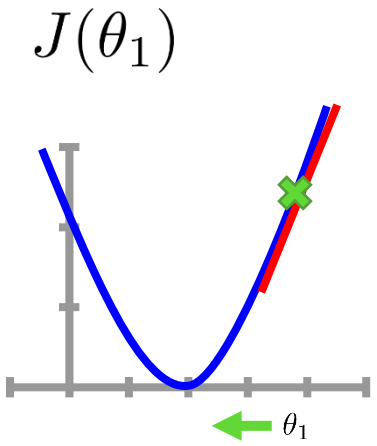
Correct: Simultaneous update

$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\theta_1 := \text{temp1}$$

Incorrect:

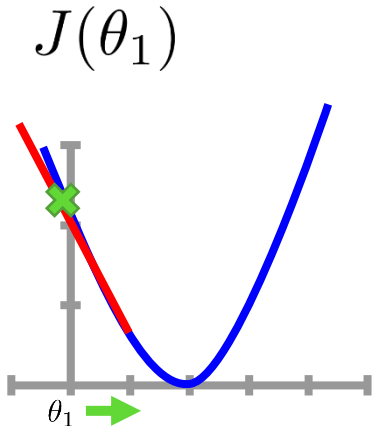
$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_1 := \text{temp1}$$

Gradient Descent Intuition



$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

$$\frac{\partial}{\partial \theta_1} J(\theta_1) > 0$$

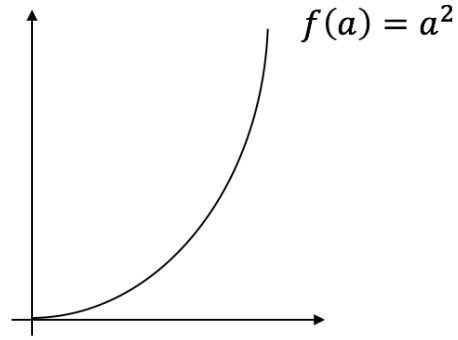


$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

$$\frac{\partial}{\partial \theta_1} J(\theta_1) < 0$$

Note: The gradient of a function represent how small changes on a parameter θ_1 will affect the cost function $J(\theta_1)$

For example:



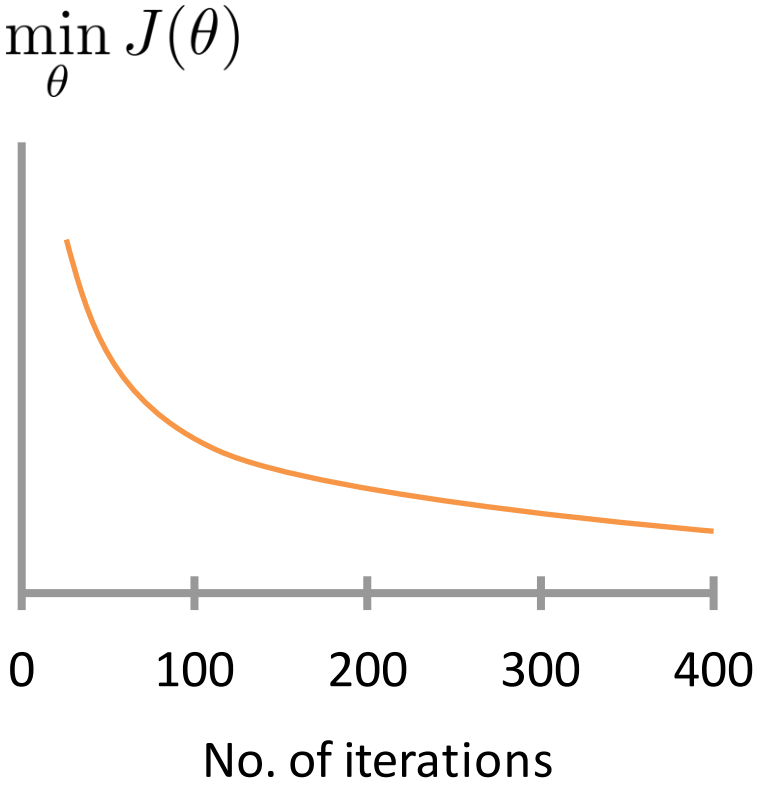
$$\frac{d\, f(a)}{da} = 2a$$

Slope (derivative) of $f(a)$ at $a = 5$ is 10

$a = 5$ $\quad f(5) = 25$

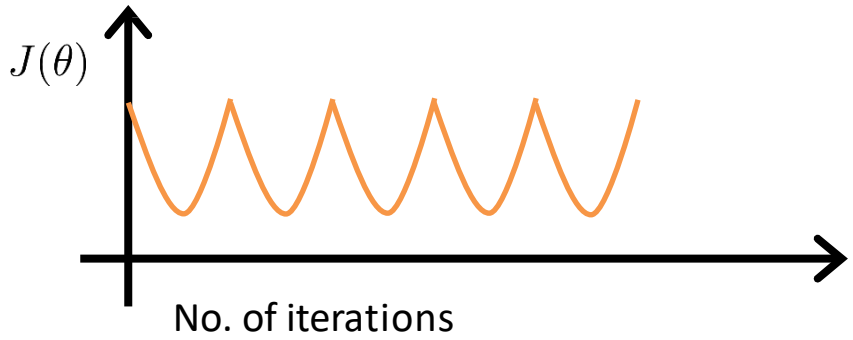
$a = 5.001$ $\quad f(5.001) \cong 25.010$

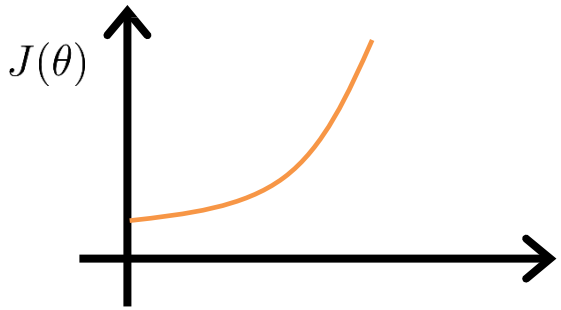
Gradient Descent working correclty?



Gradient Descent working correclty?

- For sufficiently small α, $J(\theta)$, should decrease on every iterations
- But if α is too small, gradient descent can be slow to converge
- If α is too large: $J(\theta)$ may not decrease on every iterations; may not converge
- Here some examples:

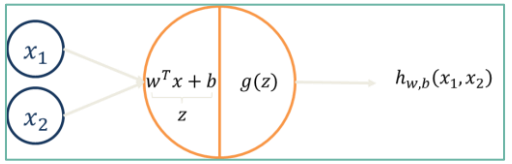




To choose α try: $\ldots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, \ldots$.

Neural Networks and Gradient Descent

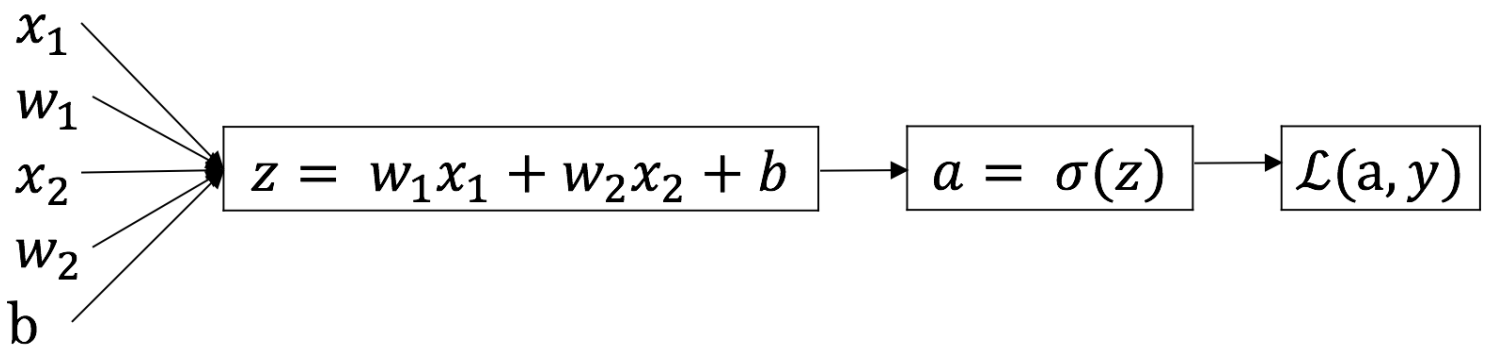
Neural Networks & Gradient Descent



Let's suppose you have the following neural network:

$z = w^T x + b$
$a = \sigma(z)\ where\ \sigma\ is\ the\ Sigmoid\ function$
$Cost(\mathrm{a}, \mathrm{y}) = \mathcal{L}(a, y) = -(yln(a) + (1-y)\ln(1-a))$

Using computational graph, we can represent as following:



Remember:

- we want to change parameters to reduce the loss.

- We can use the Gradient Descent Algorithm. Therefore, we need to compute derivatives…

Derivatives

$f(x)$	$f'(x)$	$f(x)$	$f'(x)$
x^n	nx^{n-1}	e^x	e^x
$\ln(x)$	$1/x$	$\sin(x)$	$\cos(x)$
$\cos(x)$	$-\sin(x)$	$\tan(x)$	$\sec^2(x)$
$\cot(x)$	$-\mathrm{cosec}^2(x)$	$\sec(x)$	$\sec(x)\tan(x)$
$\mathrm{cosec}(x)$	$-\mathrm{cosec}(x)\cot(x)$	$\tan^{-1}(x)$	$1/(1+x^2)$
$\sin^{-1}(x)$	$1/\sqrt{1-x^2}$ for $\lvert x\rvert < 1$	$\cos^{-1}(x)$	$-1/\sqrt{1-x^2}$ for $\lvert x\rvert < 1$
$\sinh(x)$	$\cosh(x)$	$\cosh(x)$	$\sinh(x)$
$\tanh(x)$	$\mathrm{sech}^2(x)$	$\coth(x)$	$-\mathrm{cosech}^2(x)$
$\mathrm{sech}(x)$	$-\mathrm{sech}(x)\tanh(x)$	$\mathrm{cosech}(x)$	$-\mathrm{cosech}(x)\coth(x)$
$\sinh^{-1}(x)$	$1/\sqrt{x^2+1}$	$\cosh^{-1}(x)$	$1/\sqrt{x^2-1}$ for $x > 1$
$\tanh^{-1}(x)$	$1/(1-x^2)$ for $\lvert x\rvert < 1$	$\coth^{-1}(x)$	$-1/(x^2-1)$ for $\lvert x\rvert > 1$

Derivative of Sigmoid Function

Let's denote the sigmoid function as $\sigma(x) = \frac{1}{1+e^{-x}}$

Its derivative is:

$$
\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}\left(1+\mathrm{e}^{-x}\right)^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x) \cdot (1-\sigma(x))
\end{aligned}
$$

Neural Network derivatives

$z = w^T x + b$
$a = \sigma(z)$
$Cost(\mathrm{a}, \mathrm{y}) = \mathcal{L}(a, y) = -(yln(a) + (1-y)\ln(1-a))$

Derivatives:

$$\frac{d\mathcal{L}}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{da}\frac{da}{dz} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)(a(1-a)) = a - y$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{dw_1} = (a-y)x_1$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{dw_2} = (a-y)x_2$$

$$\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{db} = (a-y)$$

Gradient Descent Algorithm for Nerual Network is based on:

$$w_1 := w_1 - \alpha \frac{d\ J(w,b)}{dw_1}$$
$$w_2 := w_2 - \alpha \frac{d\ J(w,b)}{dw_2}$$
$$b := b - \alpha \frac{d\ J(w,b)}{db}$$

Neural Networks on *m* examples

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

where:

$\mathcal{L}(a^{(i)}, y^{(i)}) = -\left(y^{(i)} ln(a^{(i)}) + (1 - y^{(i)}) \ln(1 - a^{(i)})\right)$

$a^{(i)} = \sigma(z^{(i)})$

$z^{(i)} = w^T x^{(i)} + b$

$$\frac{d\ J(w,b)}{dw_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{d\mathcal{L}(a^{(i)}, y^{(i)})}{dw_1} = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_1^{(i)}$$

$$\frac{d\ J(w,b)}{dw_2} = \frac{1}{m}\sum_{i=1}^{m} \frac{d\mathcal{L}(a^{(i)}, y^{(i)})}{dw_1} = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_2^{(i)}$$

$$\frac{d\ J(w,b)}{db} = \frac{1}{m}\sum_{i=1}^{m} \frac{d\mathcal{L}(a^{(i)}, y^{(i)})}{dw_1} = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

Gradient Descent Algorithm for Nerual Network is based on:

$$w_1 := w_1 - \alpha \frac{d\ J(w,b)}{dw_1}$$

$$w_2 := w_2 - \alpha \frac{d\ J(w,b)}{dw_2}$$

$$b := b - \alpha \frac{d\ J(w,b)}{db}$$

Remember: the sum rule for derivatives states that **the derivative of a sum is equal to the sum of the derivatives.**

Derivatives of Activation Functions

Derivatives

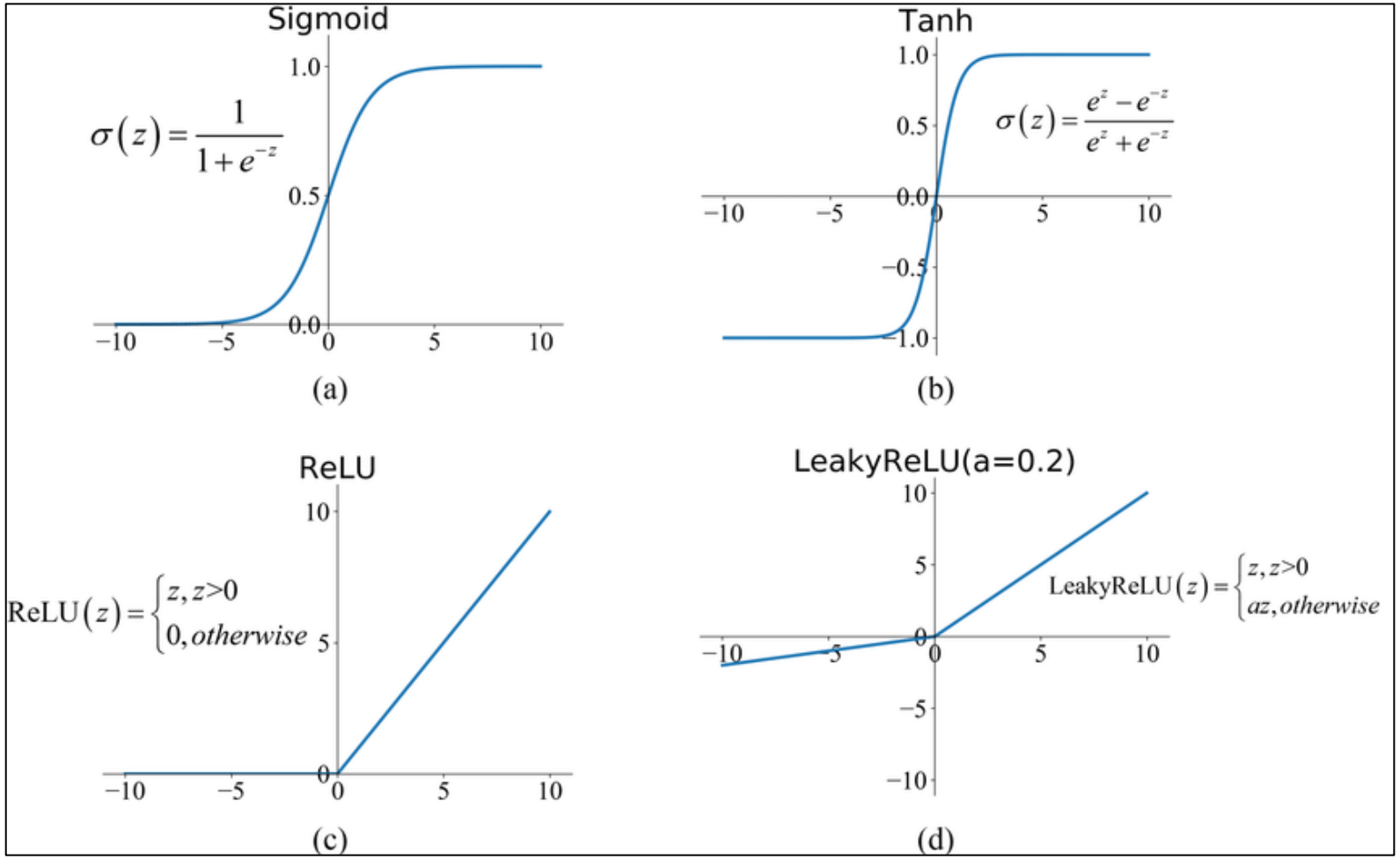
a) $\frac{d\,\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

b) $\frac{d\,\sigma(x)}{dx} = 1-\sigma(x)^2$

c) $\frac{d\,\sigma(x)}{dx} = \begin{cases} 0 & if\ z<0 \\ 1 & if\ z>0 \\ undefine\ if\ z=0^* \end{cases}$

d) $\frac{d\,\sigma(x)}{dx} = \begin{cases} a & if\ z<0 \\ 1 & if\ z>0 \\ undefine\ if\ z=0^* \end{cases}$

* No relevant in practical scenarios



Matrix – Partial Derivatives

The derivative of a vector function (a vector whose components are functions) $\boldsymbol{y} = [y_1 \;\; y_2 \; \ldots \;\; y_m]^T$, with respect to an input vector, $\boldsymbol{x} = [x_1 \;\; x_2 \;\; \ldots \;\; x_n]^T$ is written as

$$\frac{d\boldsymbol{y}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{dy_1}{dx_1} & \cdots & \frac{dy_1}{dx_n} \\ \vdots & \ddots & \vdots \\ \frac{dy_m}{dx_1} & \cdots & \frac{dy_m}{dx_n} \end{bmatrix}$$

The derivative of a vector function $\boldsymbol{y}$ with respect to an input vector, $\boldsymbol{x}$ whose components represent a space is known as the Jacobian Matrix.

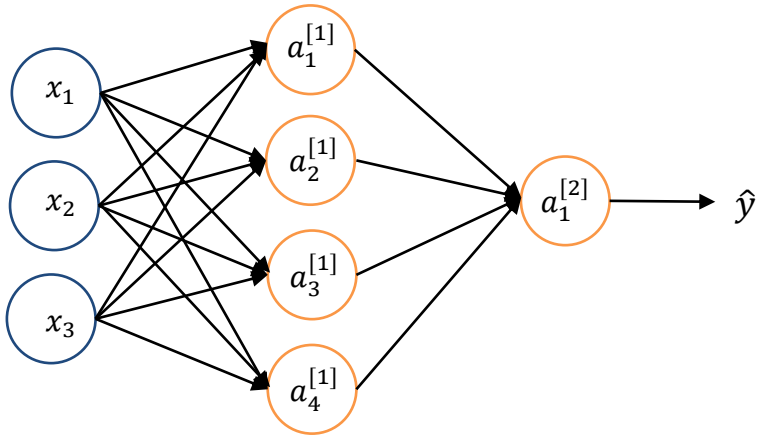
More Complex Neural Network

Let's suppose to design a Neural Network for Binary Classification

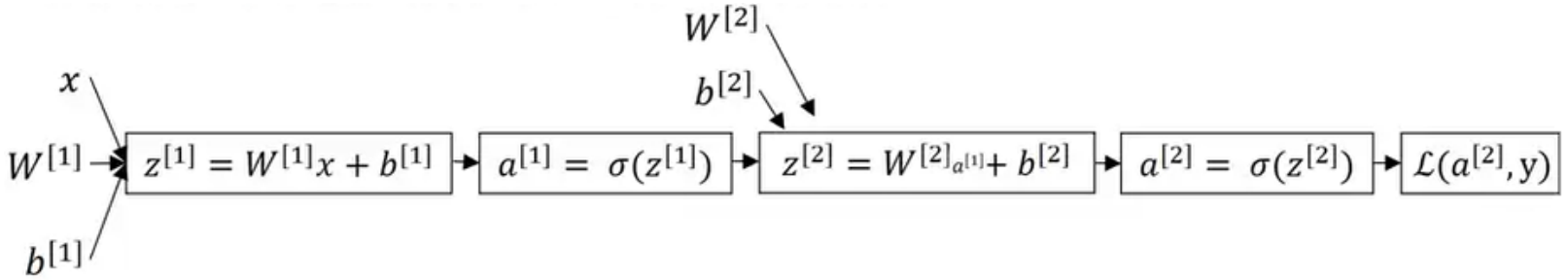
Parameters: $W^{[1]}, b^{[1]},\ W^{[2]}, b^{[2]}$

Cost Function: $\mathrm{J}\left(W^{[1]}, b^{[1]} W^{[2]}, b^{[2]}\right) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$

$\mathcal{L}(\hat{y}, y) = \mathcal{L}\left(a^{[2]}, y\right) = -(y ln\left(a^{[2]}\right) + (1-y)\ln\left(1-a^{[2]}\right))$

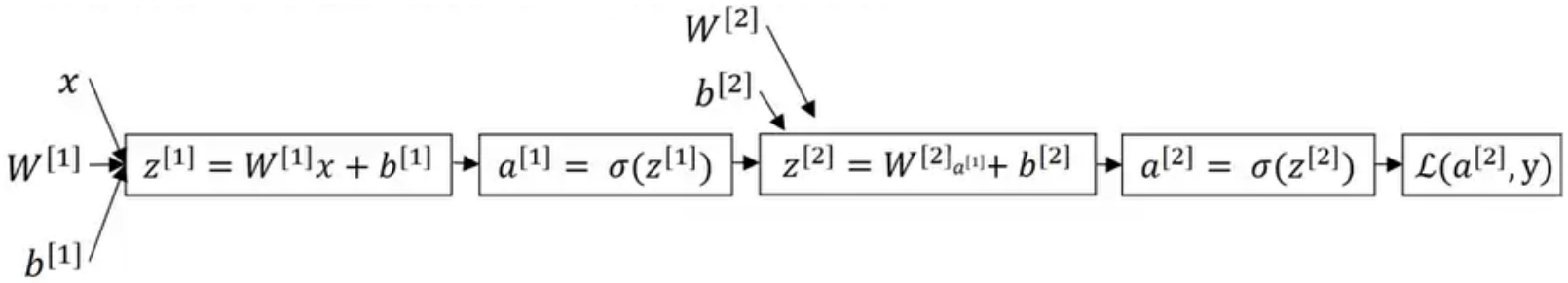


Let's start with one sample:



Remember: Learning Algorithm - Forward and Back propagation steps.

Neural Network Gradients



Forward propagation step:

$z^{[1]} = W^{[1]}x + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

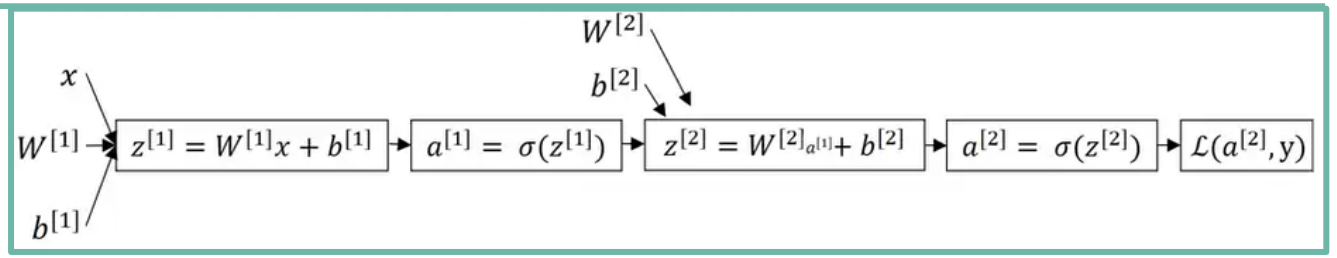
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

$a^{[2]} = g^{[2]}(z^{[2]})$

Neural Network Gradients

Loss:

$$\mathcal{L}(\hat{y}, y) = \mathcal{L}(a^{[2]}, y) = -(yln(a^{[2]}) + (1-y)\ln(1-a^{[2]}))$$



Back propagation step:

$$\frac{d\mathcal{L}}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

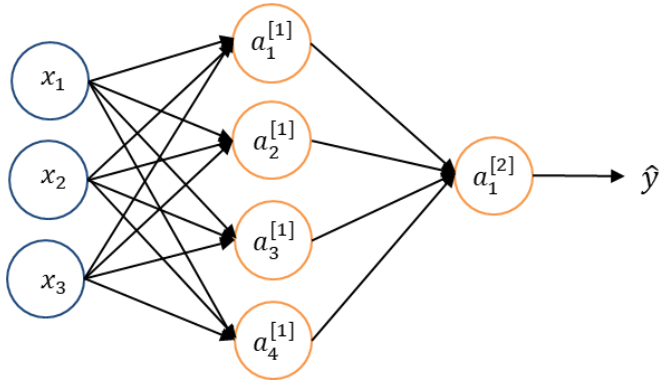
$$\frac{d\mathcal{L}}{dz^{[2]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}} = \left(-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right)\left(a^{[2]}(1-a^{[2]})\right) = a^{[2]} - y$$

$$\boldsymbol{\frac{d\mathcal{L}}{d\,W^{[2]}}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{d\,W^{[2]}} = \boldsymbol{(a^{[2]} - y)a^{[1]T}}$$

$$\boldsymbol{\frac{d\mathcal{L}}{d\,b^{[2]}}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{d\,b^{[2]}} = \boldsymbol{(a^{[2]} - y)}$$

$$\frac{d\mathcal{L}}{da^{[1]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}} = (a^{[2]} - y)W^{[2]T} = W^{[2]T}(a^{[2]} - y)$$

$$\frac{d\mathcal{L}}{d\,z^{[1]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}} = W^{[2]T}(a^{[2]} - y) * (a^{[1]} * (1-a^{[1]}))$$



Given in input X:

$z^{[1]} = W^{[1]}x + b^{[1]}$

(4,1) (4,3) (3,1) (4,1) - Dimensionality

$a^{[1]} = g^{[1]}(z^{[1]})$

(4,1) (4,1) - Dimensionality

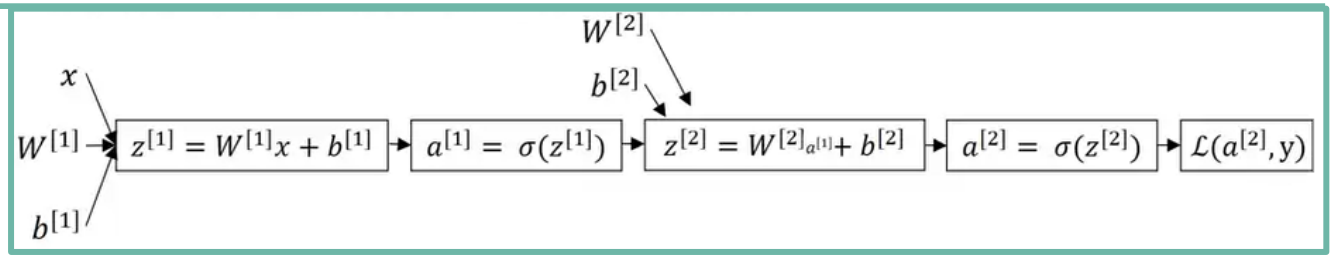
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

(1,1) (1,4) (4,1) (1,1) - Dimensionality

$a^{[2]} = g^{[2]}(z^{[2]})$

(1,1) (1,1) - Dimensionality

Neural Network Gradients



Back propagation step:

$$\frac{d\mathcal{L}}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

$$\frac{d\mathcal{L}}{dz^{[2]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}} = \left(-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right)\left(a^{[2]}(1-a^{[2]})\right) = a^{[2]} - y$$

$$\boldsymbol{\frac{d\mathcal{L}}{d\,W^{[2]}}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{d\,W^{[2]}} = \boldsymbol{(a^{[2]} - y)a^{[1]T}}$$

$$\boldsymbol{\frac{d\mathcal{L}}{d\,b^{[2]}}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{d\,b^{[2]}} = \boldsymbol{(a^{[2]} - y)}$$

$$\frac{d\mathcal{L}}{da^{[1]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}} = (a^{[2]} - y)W^{[2]T} = W^{[2]T}(a^{[2]} - y)$$

$$\frac{d\mathcal{L}}{d\,z^{[1]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}} = W^{[2]T}(a^{[2]} - y) * (a^{[1]} * (1 - a^{[1]}))$$

$$\boldsymbol{\frac{d\mathcal{L}}{d\,W^{[1]}}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}}\frac{dz^{[1]}}{dW^{[1]}} = \frac{d\mathcal{L}}{d\,z^{[1]}}x^T$$

$$\boldsymbol{\frac{d\mathcal{L}}{d\,b^{[1]}}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}}\frac{dz^{[1]}}{db^{[1]}} = \frac{d\mathcal{L}}{d\,z^{[1]}}$$

Gradient Descent Algorithm :

$$W^{[1]} := W^{[1]} - \alpha\frac{d\,\mathrm{J}(W^{[1]}, b^{[1]}W^{[2]}, b^{[2]})}{dW^{[1]}}$$

….

Neural Network Gradients with *m* examples

$$\mathrm{J}\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(a^{[2](i)}, y^{(i)}\right)$$

Back propagation step with m examples:

$$\frac{d\,J\left(W^{[1]}, b^{[1]}W^{[2]}, b^{[2]}\right)}{dW^{[2]}} = \frac{1}{m}\sum_{i=1}^{m}\frac{d\,\mathcal{L}\left(a^{[2](i)}, y^{(i)}\right)}{d\,W^{[2]}}$$

$$\frac{d\,J\left(W^{[1]}, b^{[1]}W^{[2]}, b^{[2]}\right)}{db^{[2]}} = \frac{1}{m}\sum_{i=1}^{m}\frac{d\,\mathcal{L}\left(a^{[2](i)}, y^{(i)}\right)}{d\,b^{[2]}}$$

$$\frac{d\,J\left(W^{[1]}, b^{[1]}W^{[2]}, b^{[2]}\right)}{dW^{[1]}} = \frac{1}{m}\sum_{i=1}^{m}\frac{d\,\mathcal{L}\left(a^{[2](i)}, y^{(i)}\right)}{d\,W^{[1]}}$$

$$\frac{d\,J\left(W^{[1]}, b^{[1]}W^{[2]}, b^{[2]}\right)}{db^{[1]}} = \frac{1}{m}\sum_{i=1}^{m}\frac{d\,\mathcal{L}\left(a^{[2](i)}, y^{(i)}\right)}{d\,b^{[1]}}$$

Gradient Descent Algorithm:

$$W^{[2]} := W^{[2]} - \alpha\frac{d\,\mathrm{J}\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right)}{dW^{[2]}}$$

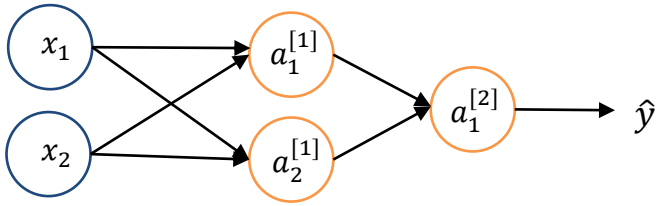
$$b^{[2]} := b^{[2]} - \alpha\frac{d\,\mathrm{J}\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right)}{db^{[2]}}$$

...

Random Initialization

What happens if you initialize weights to zero?

Let's start with this example with Sigmoid function as activation functions:



Derivatives from previous slides:

$$\frac{d\mathcal{L}}{d\,W^{[2]}} = (a^{[2]} - y)a^{[1]T}$$

$$\frac{d\mathcal{L}}{d\,z^{[1]}} = W^{[2]T}(a^{[2]} - y) * (a^{[1]} * (1 - a^{[1]}))$$

$$\frac{d\mathcal{L}}{d\,W^{[1]}} = \frac{d\mathcal{L}}{d\,z^{[1]}}x^T$$

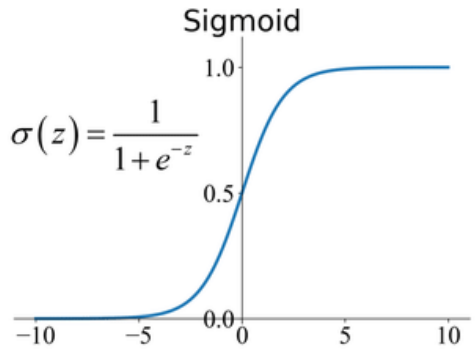
If $W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and, $W^{[2]} = [0 \quad 0]$ for any input sample in forward propagation $a_1^{[1]} = a_2^{[1]} \rightarrow \frac{d\mathcal{L}}{dW_1^{[2]}} = \frac{d\mathcal{L}}{dW_2^{[2]}} \rightarrow \frac{d\mathcal{L}}{dz_1^{[1]}} = \frac{d\mathcal{L}}{dz_2^{[1]}}$

Let's suppose $\frac{d\mathcal{L}}{dz_1^{[1]}} = \frac{d\mathcal{L}}{dz_2^{[1]}} = v$, when we compute the $\frac{d\mathcal{L}}{d\,W^{[1]}}$ we can see that:

$$\frac{d\mathcal{L}}{d\,W^{[1]}} = \frac{d\mathcal{L}}{d\,z^{[1]}}x^T = \begin{bmatrix} v \\ v \end{bmatrix}[x_1 \quad x_2] = \begin{bmatrix} vx_1 & vx_2 \\ vx_1 & vx_2 \end{bmatrix}$$

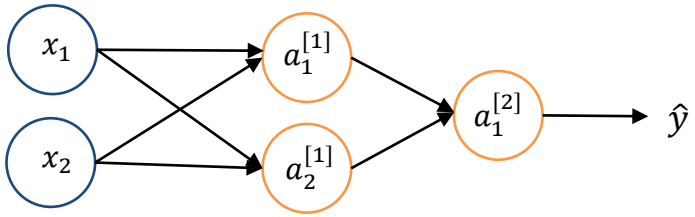
Thus, in the second forward interaction computing we will have $a_1^{[1]} = a_2^{[1]}$ and so on....
It's called "symmetry problem"

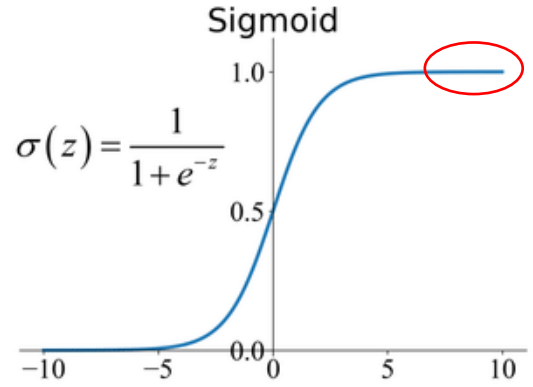
Random initialization is a solution! (Note: biases initialization to zero is not a problem)



Random Initialization

Let's start with this example:





$W^{[1]} = np.radom.randn\big((2,2)\big) * \gamma$

$b^{[1]} = np.zeros\big((2,1)\big)$

$W^{[2]} = np.radom.randn\big((1,2)\big) * \gamma$

$b^{[2]} = 0$

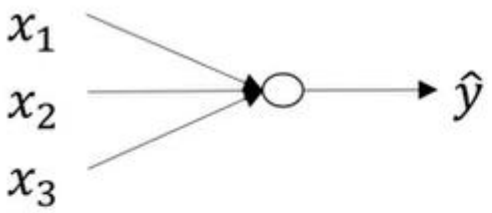
For example, $\gamma = 0.01$

The main idea is to try to keep values small!

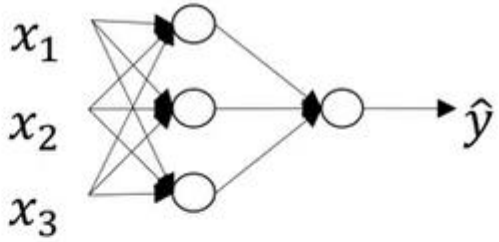
Otherwise, if you use **large weights**, for example with a **sigmoid function** in the last layer, your output values will go on the flat region of the function where **the slop or the gradient is very small**. **Gradient Descent will be very slow**.

What is a Deep Neural Network?

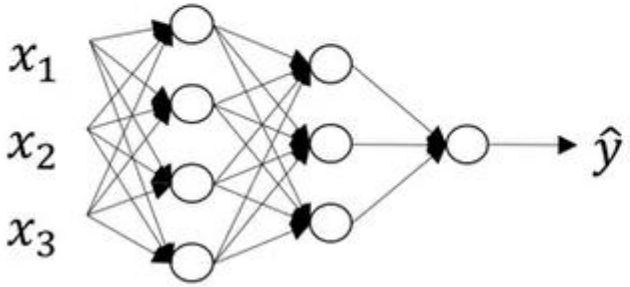
«Shallow Network»



logistic regression

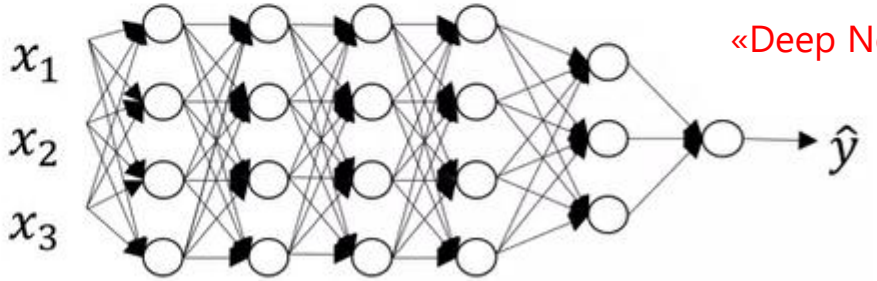


1 hidden layer



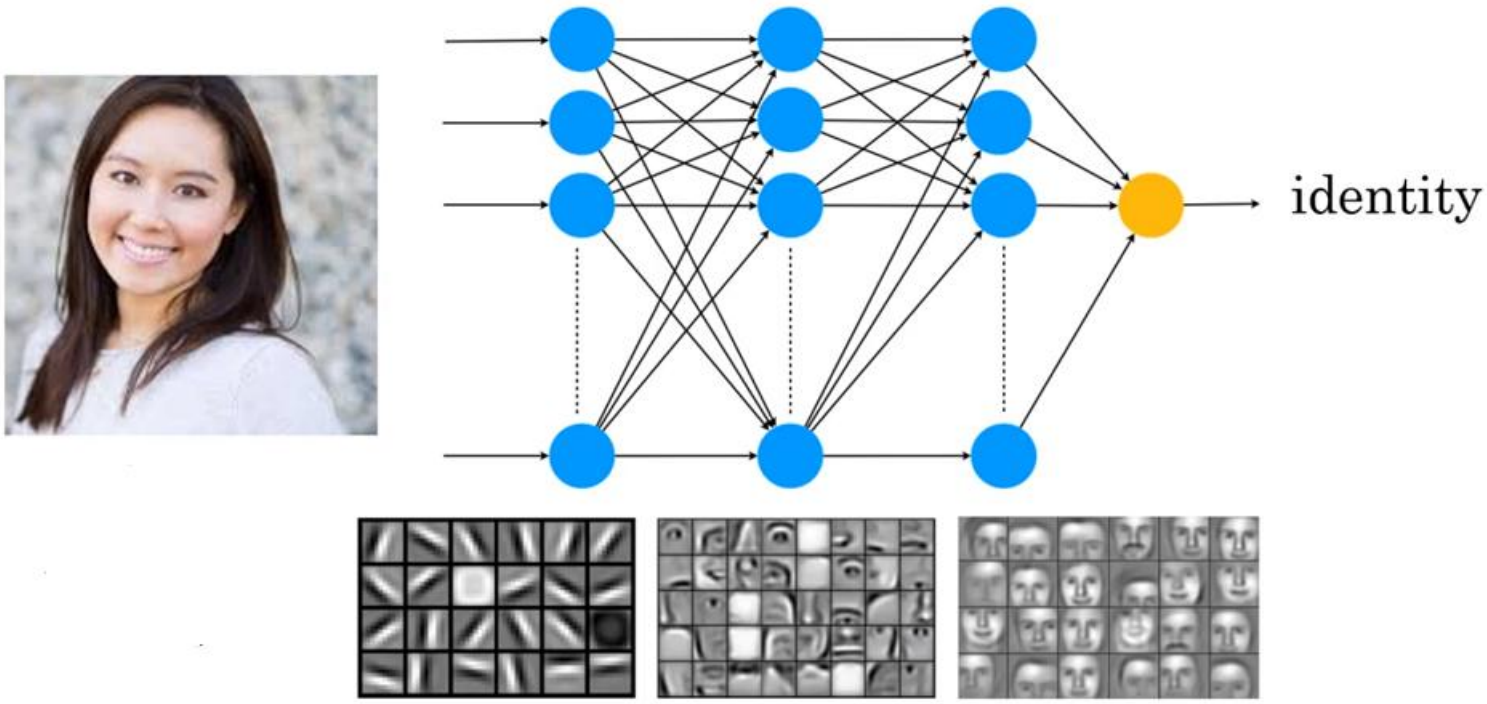
2 hidden layers

«Deep Network»



5 hidden layers

Intuition about deep representation



We will better understand this image when we are using Convolution Neural Networks.

The idea could be applied to audio context: phonemes… words… sentences

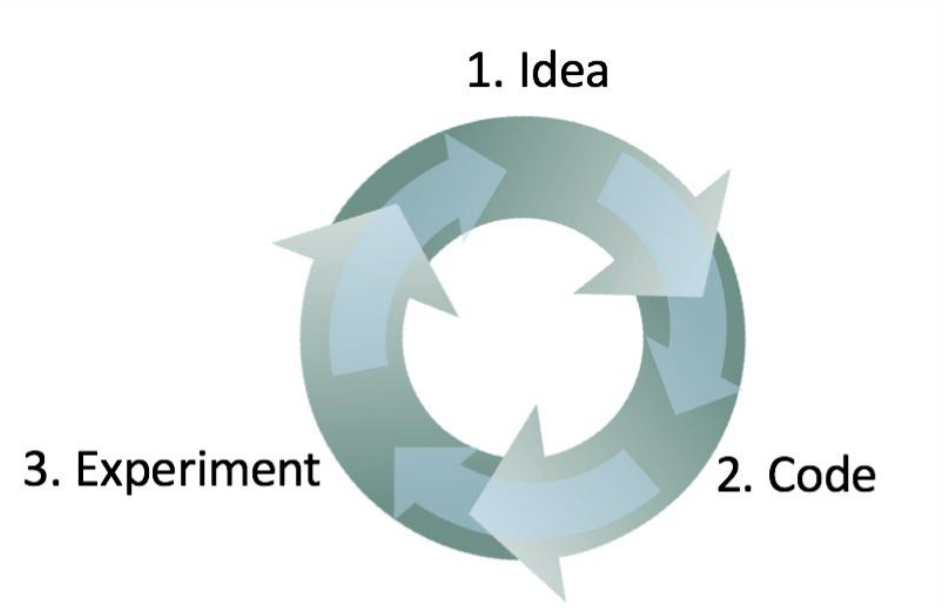
What are hyperparameters?

Parameters: $W^{[1]}, b^{[1]} W^{[2]}, b^{[2]}$....

Hyperparameters:

- Learning rate
- Number of interaction (epochs)
- Number of hidden layers
- Number of hidden units in each hidden layers
- Activation functions

Right now, this process is empirically.



Advanced Loss Functions

Loss functions

For Binary Classification we often use the **Binary Cross-Entropy Loss**:

$$\mathrm{L}(\mathrm{W},\mathrm{b}) = -\frac{1}{m}\sum_{i=1}^{m} -y^{(i)} \log\left(h_{W,b}\left(x^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-h_{W,b}\left(x^{(i)}\right)\right)$$

Where m is the number of examples in the mini-batch (see also Cost Function of the Logistic Regression)

For Regression we often use the **Mean Squared Error Loss**:

$$\mathrm{L}(\mathrm{W},\mathrm{b}) = \frac{1}{m}\sum_{i=1}^{m}\left(h_{W,b}\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Where m is the number of examples in the mini-batch

Note: in the regression setting the network ends with a single unit and no activation. Thus, the last layer is purely “linear”, and the network is free to learn to predict values in any range.

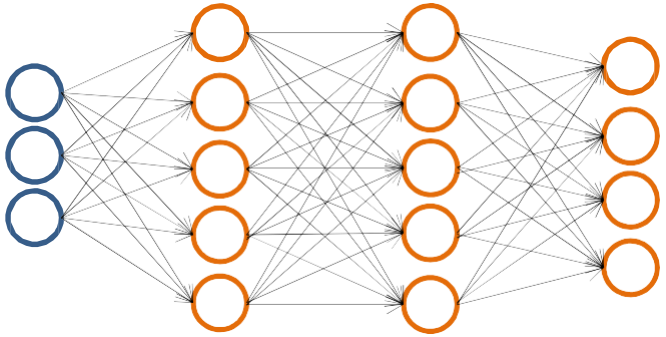
Multiclass Classification

Pedestrian

Car

Motorcycle

Truck



Multiclass Classification: One-hot Representation

Pedestrian

Car

Motorcycle

Truck

One-hot representation:

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix} \quad \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \quad \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \quad \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

SoftMax Layer/Activation

In multiclass classification the last layer of a neural network we often use a Softmax activation.

It means the network will output a *probability distribution* over the classes

Formally, let's $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$

Softmax layer is:

$$a^{[i]} = \frac{e^{z^{[i]}}}{\sum_{J=1}^{N} e^{z_j^{[i]}}}$$ where N is the numbe of classes

Example:

$$z^{[i]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \rightarrow a^{[i]} = \begin{bmatrix} \frac{e^5}{t} \\ \frac{e^2}{t} \\ \frac{e^{-1}}{t} \\ \frac{e^3}{t} \end{bmatrix} \text{ where } t = \left(e^5 + e^2 + e^{-1} + e^3\right) \rightarrow a^{[i]} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$ The Sum is equal to 1

Cross-Entropy Loss

To train a neural network for a multiclass problem we can use the **Cross-Entropy Loss**. Cross-entropy is commonly used to quantify the difference between two probability distributions.

Usually the "true" distribution (the one that your machine learning algorithm is trying to predict) is expressed in terms of a one-hot distribution.

$$\mathrm{L}(\mathrm{W},\mathrm{b}) = -\sum_{i=1}^{m} y^{(i)} \log\left(h_{W,b}\left(x^{(i)}\right)\right)$$

Where m is the number of training samples, $y^{(i)}$ is the ground truth output distribution and $h_{W,b}\left(x^{(i)}\right)$ is the predicted output distribution.

In practice, $y^{(i)}$ is a one-hot representation of the right label.

Cross-Entropy Loss

For Example:

$y^{(i)} = [0\ 1\ 0]^T; h_{W,b}(x^{(i)}) = [0.228\ 0.619\ 0.153]^T$

$L = -(0 * \ln(0.228) + 1 * \ln(0.619) + 0 * \ln(0.153)) = 0.479$

The **Cross-Entropy Loss focuses on the positive class**, because **the negative classes are potentially infinite**.

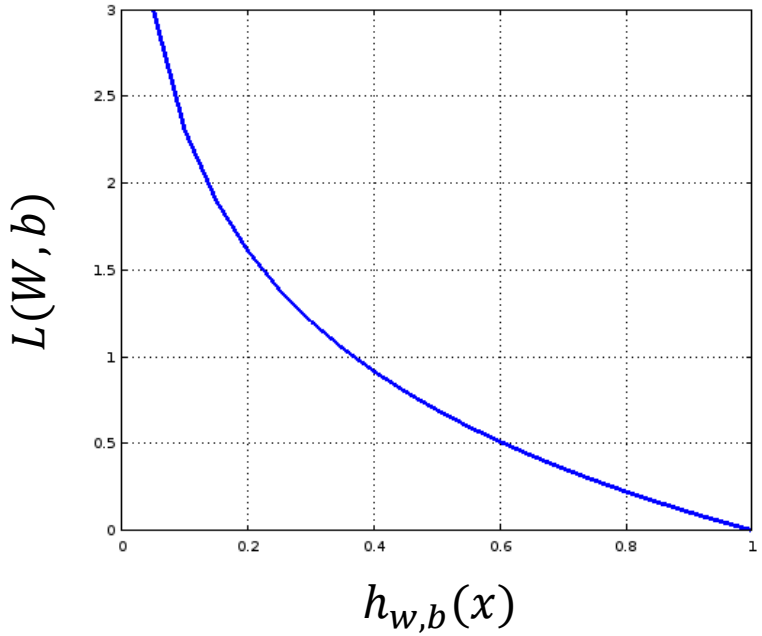
Another Example:

Exactly right: $L = -(1 * \ln(1)) = 0$

50% Probability on correct target: $L = -(1 * \ln(0.5)) = 0.693$

25% Probability on correct target: $L = -(1 * \ln(0.25)) = 1.386$

0% Probability on correct target: $L = -(1 * \ln(0)) = \infty$



Ranking Loss Function

Unlike other loss functions, such as Cross-Entropy Loss or Mean Square Error Loss, whose objective is to learn to predict directly a label, a value, or a set or values given an input, **the objective of Ranking Losses is to predict relative distances between inputs**. This task if often called **metric learning**.

Ranking Losses functions are very flexible in terms of training data:

- **We just need a similarity score between data points to use them**.
- That score can be binary (similar / dissimilar)

For example:

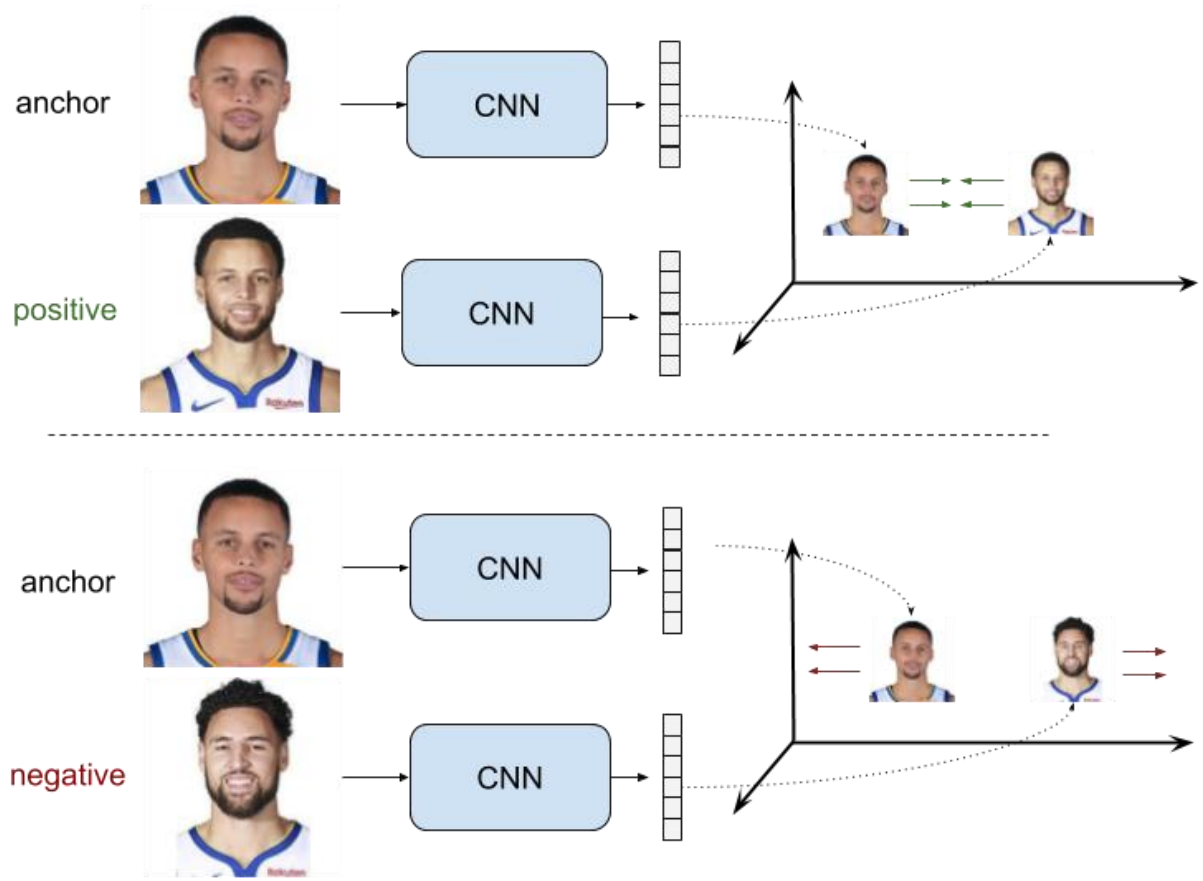
Imagine a face verification dataset, where **we know which face images belong to the same person** (similar), and which not (dissimilar). Using a Ranking Loss function, we can train a NN (or a CNN) to infer if two face images belong to the same person or not.

Pairwise Ranking Loss

In this setup positive and negative pairs of training data points are used.

Positive pairs are composed by an anchor sample x_a and a positive sample x_p, which is similar to x_a in the metric we aim to learn

Negative pairs composed by an anchor sample x_a and a negative sample x_n, which is dissimilar to x_n in that metric.



Pairwise Ranking Loss

The objective is to learn representations with a small distance d between them for positive pairs, and greater distance than some margin value m for negative pairs.

Pairwise Ranking Loss forces representations to have zero distance for positive pairs, and a distance greater than a margin for negative pairs.

Being r_a, r_p and r_n the samples representations and d a distance function, we can write:

$$L = \begin{cases} d(r_a, r_p) & if\ Positive\ Pair \\ max(0, m - d(r_a, r_n)) & if\ Negative\ Pair \end{cases}$$

For positive pairs, **the loss will be 0 only when the net produces representations for both the two elements in the pair with no distance between them**

Pairwise Ranking Loss

$$\begin{cases} d(r_a, r_p) & if\ Positive\ Pair \\ max(0, m - d(r_a, r_n)) & if\ Negative\ Pair \end{cases}$$

For negative pairs, the loss will be 0 when the distance between the representations of the two pair elements is greater than the margin m.

When that distance is not bigger than m, the loss will be positive, and net parameters will be updated to produce more distant representation for those two elements.

The loss value will be at most m, when the distance between r_a and r_n is 0.

The function of the margin is that, when the representations produced for a negative pair are distant enough, no efforts are wasted on enlarging that distance.

If r_0 and r_1 are the pair elements representations, y is a binary flag equal to 0 for a negative pair and to 1 for a positive pair and the distance dd is the Euclidian distance, we can equivalently write:

$$L(r_0, r_1) = y\|r_0 - r_1\| + (1 - y)max(0, m - \|r_0 - r_1\|)$$